AFRL-IF-RS-TR-2004-334
Final Technical Report
December 2004

COMPOSABLE FORMAL MODELS FOR HIGH-ASSURANCE FAULT TOLERANT NETWORKS

SRI International

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. N442

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

STINFO FINAL REPORT

AFRL-IF-RS-TR-2004-334 has been reviewed and is approved for publication

APPROVED: /s/
DAVID E. KRZYSIAK
Project Engineer

FOR THE DIRECTOR: /s/
WARREN H. DEBANY, JR.
Technical Advisor
Information Grid Division
Information Directorate

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 074-0188*

1. AGENCY USE ONLY (Leave blank)	**2. REPORT DATE** December 2004	**3. REPORT TYPE AND DATES COVERED** **FINAL** **Jun 02 – Jun 04**

4. TITLE AND SUBTITLE COMPOSABLE FORMAL MODELS FOR HIGH-ASSURANCE FAULT TOLERANT NETWORKS	**5. FUNDING NUMBERS** C - F30602-02-C-0130 PE - 62301E PR - N442 TA - FT WU - N1
6. AUTHOR(S) Carolyn Talcott	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) SRI International 333 Ravenswood Avenue Menlo Park CA 94025-3493	**8. PERFORMING ORGANIZATION REPORT NUMBER** N/A
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Defense Advanced Research Projects Agency, 3701 North Fairfax Drive, Arlington VA 22203-1714 AFRL/IFGA, 525 Brooks Road, Rome NY 13441-4505	**10. SPONSORING / MONITORING AGENCY REPORT NUMBER** AFRL-IF-RS-TR-2004-334

11. SUPPLEMENTARY NOTES

AFRL Project Engineer: David E. Krzysiak/IFGA/(315) 330-7454 David.Krzysiak@rl.af.mil

12a. DISTRIBUTION / AVAILABILITY STATEMENT *APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.*	**12b. DISTRIBUTION CODE**

13. ABSTRACT *(Maximum 200 Words)*

This effort carried out substantial case studies involving analysis of different network services in order to develop modeling and analysis methodologies and libraries of reusable models to aid in achieving higher assurance for and more robust designs of network systems. There are four main results: 1) an analysis of a java secure proxy toolkit with models of attacks, mitigations, and patterns; 2) a modular formal executable model of the secure spread group communications system; 3) formal models of Distributed Denial of Service (DDoS) attacks and mitigation services complementing OPNET simulations; and 4) a first prototype of Mobile Maude.

14. SUBJECT TERMS network services case studies, Distributed Denial of Service (DDoS) attacks, DDoS attack mitigation, formal models	**15. NUMBER OF PAGES** 17
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Table of Contents

List of Figures

1.0 Introduction and Overview

A two-year project received funding of $999, 739, out of a budgeted ceiling of $1,199,739. Senior personnel were Dr. Carolyn Talcott (PI) and Prof. Jos´e Meseguer. Other team members included Dr. Steven Eker, Dr. Mark-Oliver Stehr, and Ambarish Sridharanarayanan. Driven by a series of substantial case studies, the project aimed to develop modeling and analysis methodologies and libraries of reusable models to enable network systems to achieve higher assurance and more robust designs.

Several candidates for case studies were identified among systems being developed as part of the Fault Tolerant Networks (FTN) and Dynamic Coalitions (DC) DARPA programs. After preliminary investigations, three were selected for detailed formal modeling and analysis: the Java Secure Proxy Toolkit (Stanford-SRI DC project), Secure Spread (The Johns Hopkins University (JHU) DC project), and Distribute Denial of Service models (JHU APL FTN project). An additional task was the development of a prototype of Mobile Maude. The formal models, analyses, documentation, and slide presentations developed under this contract are available from the project Web site at http://www-formal.stanford.edu/clt/FTN.

The models were developed using the rewriting logic language Maude. Rewriting logic [11, 13] is a simple logic well suited for distributed system specification, that is executable and reflective (capable of faithfully representing important aspects of its own syntax and deductive/computation mechanisms [8]). The Maude system is an implementation of rewriting logic and its reflective capabilities. Maude 2.0 was released in June 2003, with a system presentation at the Rewriting Techniques and Applications conference in Valencia. In addition to many efficiency improvements in the rewriting engine, Maude 2.0 implements an object and message fair rewriting strategy that is important for the development of Mobile Maude. Maude 2.1 was released in March 2004, with a system presentation at the Workshop on Rewriting Logic and Applications in Barcelona. Maude 2.1 provides several new reflective capabilities and operations for module composition and renaming. As of version 2.0 Maude is open source and the source tree, binaries for several platforms, documentation, examples, and papers are available on the Maude Web site at maude.cs.uiuc.edu (mirrored at maude.csl.sri.com).

The products of the case studies include:

- Executable attack models for Java's proxy-based remote service
- Abstract models of crypto libraries (supporting the CLIQUES API)
- Modular executable models of the Spread group communication system and its extensions to support virtual synchrony and secure communication—these models were designed to be easy to modify in order to support prototyping and analysis of alternative algorithm design decisions
- New compositional modeling and formal analysis techniques
- Models of Distributed Denial-of-Service (DDOS) attacks and mitigation technologies
- Maude modules defining the prototype Mobile Maude infrastructure and examples of mobile agent code

The three case studies and Mobile Maude prototype are summarized below. Full details are presented in supporting documents (available from the project web site).

2.0 Java Secure Proxy Toolkit (SPTK)

The Secure Proxy Toolkit (SPTK), based on Java RMI and Jini technologies, was developed by John Mitchell, Ninghui Li, and Derrick Tong as part of the Stanford-SRI Dynamic Coalitions project *Agile Management of Dynamic Collaboration*. The objective of this case study was to formally model and analyze the SPTK. An additional objective was to use this case study as an example of how to develop and organize models of such systems, to model different attacks, and to analyze the service models for them in the context of possible attacks. The resulting models and analyses are available at www-formal.stanford.edu/clt/FTN/SPTK/index.html along with a tutorial document describing the models and analyses in some detail. Here we summarize this work.

In a distributed system in which servers wish to make services available to remote clients, the objective of a service proxy mechanism is to facilitate client-server interaction by providing:

1. Mechanisms for service registration and lookup,
2. Proxies that make the communication appear local to both the client and the server processes.

A secure proxy toolkit (SPTK) should satisfy the above requirement and also should transparently support a variety of security properties: protecting information communicated between client and server, assuring client and server of each other's identity, and enforcing access policies. Thus, it should provide fixed interfaces to client and server applications. Only the internal interactions change in order to provide protection against given threat conditions or meet different security needs.

The design of the SPTK was motivated by the following security goals:

I --Proxy access to client JVM and resources is controlled—this can be addressed by Java security mechanisms.

II --Communications between proxies and services are secure—this can be addressed by proxies that use secure remote communication such as SSL, but can the client rely on the proxy it receives being such a proxy?

III --Clients should be able to authenticate proxies—both code and data.

IV --Some applications require services to authenticate and authorize client access, supporting

- authentication only once in a session (single signon)
- multiple authentication mechanisms
- authorization based on arguments of calls.

The first step in developing the formal models was to define event diagrams characterizing the underlying protocols and main scenarios for use of the toolkit. This was done in collaboration with the Stanford group. A key issue was determining what properties to check. The starting point was the informal list of desired properties given by John Mitchell in his project presentation. These properties concerned mutual authentication of client and server, independent of the (possibly hostile) behavior of Registry/Lookup services. In the process of developing the event diagrams an omission in the toolkit design was discovered that allowed a client to authenticate a signed proxy for a wrong service. This has been corrected in both the implementation and the formal model.

A modular series of formal executable models was developed using Maude. Each model in the series provides an additional level of security protection of client and server, ranging from no protection to signed proxies and authenticated secure session communication. The latter models the Secure Proxy Java Toolkit developed by Stanford.

Level 0 is for use in a situation with no attacker. The job of the client and server side proxies is simply to make interaction appear local to the application and service, respectively. This level achieves security goal I by relying on the underlying JVM to protect the host system.

Level 1 provides protection against an attacker that can observe and modify communications between the client and server. Such an attacker aims to obtain a client's private information and might also modify service calls and replies. The level 1 proxies communicate using secure connections (for example SSL). This level achieves security goal II.

Level 2 provides protection against an attacker that can observe and modify communication between the lookup service and client or server as well as client server communication. To foil the attacker, the level 2 server toolkit signs the proxy, before registration; thus, modification can be detected. The client-side toolkit checks whether a proxy obtained by lookup was registered by a trusted server by checking the signature.

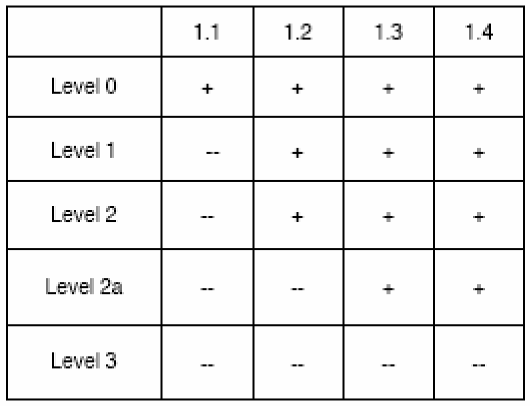
Level 2a includes a service description contained in the registered proxy. The client-side toolkit additionally confirms that the proxy received has an acceptable description. This level achieves security goal III.

Level 3 adds client authentication to ensure that the requests are from the claimed client and that they are allowed for this client. This prevents an attacker from impersonating a client, thus possibly corrupting the server's data or obtaining client secrets that result from queries containing only public data. The toolkit also sets up a secure session between the mutually authenticated server and client. This level achieves security goal IV.

Two attack models were developed: attacker-in-the-ether and compromised registry. The attacker-in-the-ether has control over the network and is free to modify or generate messages. (Deleting messages is not considered as we are not protecting against denial-of-service attacks.) A compromised registry may reply to lookup requests with any proxy it chooses.

The toolkit models were analyzed in composition with attackers. Figures 1 and 2 summarize the analysis results for each level and attack model. In these summaries, columns are labeled by the property checked (described in the accompanying key) and the rows correspond to SPTK models providing the different security levels. The symbol + means the attacker succeeds, while – means that the attacker fails.

The attacks demonstrate the need for not only checking signatures, but also that the proxy represents the requested service.



	1.1	1.2	1.3	1.4
Level 0	+	+	+	+
Level 1	--	+	+	+
Level 2	--	+	+	+
Level 2a	--	--	+	+
Level 3	--	--	--	--

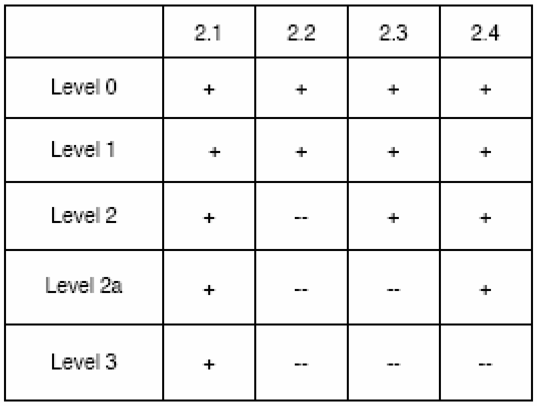
Figure 1: Analysis Results for Attacker in the Network

1.1 attacker can see/modify client data sent in service calls and replies
1.2 client accepts wrong proxy
1.3 unauthorized service call succeeds
1.4 imposter succeeds in forging client id

Figure 1 summarizes analyses for the “attacker in the network” attack model, while Figure 2 summarizes analyses for the “compromised registry” attack model.

In both cases, we see that all the attacks succeed when the level 0 toolkit is used, while none succeed when the level 3 toolkit is used. For attacks on client-server communication (1.1), level 1 protection is sufficient. In the case of the compromised lookup service, we do not worry about client-server communication, but we do sanity check to see if it is possible for the client to find and use the requested service; here, the + means that the client can succeed in all cases. For attacks on communication with the registry (1.2), level 2 protection is sufficient if both the server signature and the service description are checked (level 2a).

The distinction is made more precise in the compromised registry model where level 2f is sufficient protection to keep the client from accepting a proxy to a service provided by an untrusted server (property 2.2), while level 2a is required to assure that a proxy to the correct service is accepted (2.3). Properties 1.3, 1.4, and 2.4 deal with situations in which the server is tricked into serving improper requests. These attacks require level 3 protection.



	2.1	2.2	2.3	2.4
Level 0	+	+	+	+
Level 1	+	+	+	+
Level 2	+	--	+	+
Level 2a	+	--	--	+
Level 3	+	--	--	--

Figure 2: Analysis Results for Compromised Registry

2.1 client can obtain proxy for requested service (sanity check)
2.2 client accepts proxy to attacker service
2.3 client accepts wrong trusted server proxy
2.4 service integrity violated

3.0 Distributed Denial-of-Service Attacks

One objective is to add formal aspects to the model verification and validation done for DDoS attack models studied as part of a Johns Hopkins University Applied Physics Laboratory (JHUAPL) FTN project. The first issue is to determine which attacks and mitigation technologies are most amenable to the Maude modeling and analysis capabilities. In discussion with the JHUAPL team we determined to start with attacks for which classification mitigation technologies exist. Our first effort will be to model and analyze the TCP SYN flood attack and *Synkill* active monitoring technology recently studied by JHUAPL. Looking for misclassifications is of interest.

Network simulators can be used as a test bed for analyzing networks where implementation is infeasible either because of the premature nature of the ideas, or because of resource constraints. Maude's capability as a programming language and as a powerful formal analysis tool has helped us in implementing a prototype network simulator in Maude. The current version has basic support for discrete time, multiple nodes, multiple layered networks, and routing. A mechanism is in place for generating events (such as requests, packets) statistically.

Using this infrastructure, and in cooperation with Donna Gregg's group at JHU-APL, we have simulated a DoS attack on a 70-node low-connectivity network. In addition, a counterstrategy for DoS attacks proposed by Gene Spafford's group at Purdue, known as *synkill* has also been formally specified as a test case. Using Maude's formal analysis mechanisms, we have found some potential problems in the working of the *synkill* algorithm, where the algorithm misclassifies certain malicious hosts as benign.

4.0 Secure Group Communication

The aim of this case study was to provide an executable formal model of the services provided by the Secure Spread group communication system (a Johns Hopkins University CS Department DC project, http://www.cnds.jhu.edu/ research/group/ secure_spread/), and to develop mechanisms for model validation and lightweight analysis. The basic service a group communication system (GCS) provides is multicast of messages to groups that it manages. A GCS accepts send requests from applications and delivers multicast messages received. In addition to send and delivery events, there are view installation events delivering view messages to application processes. A view has an identifier, a set of member processes (the current members), and a set of transitional processes (those coming from the same view as the receiving process). GCS events are partially ordered according to arrival order at a given process, and send precedes a delivery causal order across processes. A message send or delivery event at a process is said to be 'in' the view whose installation at the process most recently precedes this event.

A number of properties should hold for a GCS (see [5, 14] for details). For example, Self Inclusion: if a process installs a view, it is a member of that view.Sending View Delivery: a message is delivered in the view in which it was sent. Self Delivery: if a process sends a message, it will deliver that message unless it crashes.

Virtual Synchrony: if two processes move together from one view to the next, they deliver the same messages in the first view. We refer to these as virtual synchrony (VS) properties. Spread implements an *extended* virtual synchrony (EVS) semantics that weakens the Sending View Delivery property to Same View Delivery: if two processes deliver a message, they both deliver it in the same view.

In addition, a GCS may provide different types of message delivery service with different reliability and ordering guarantees: reliable, fifo, causally ordered, totally ordered (also called agreed), or safe, where the latter means that messages are delivered only if it is known that everybody in the group has actually received it.

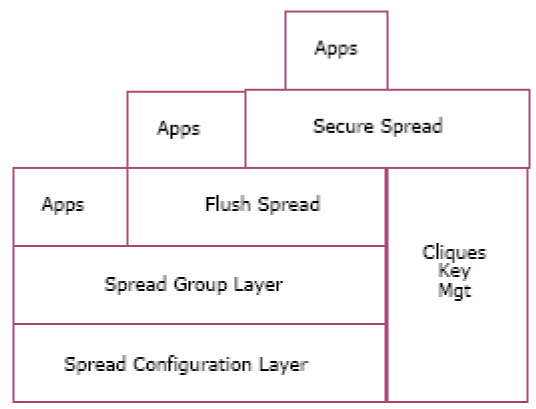


Figure 3: Secure Spread Architecture

The modular architecture of Secure Spread is shown in Figure 3. The implementation of Spread is based on two-level architecture: a configuration layer that manages the physical group of processors (which can be partitioned depending on the network connectivity) and a group layer that manages logical groups of agents. From discussions with the Spread team we determined that [2] is intended as the specification of the configuration layer and [14] should be regarded as the specification of the group layer. Both the configuration layer and the group layer provide extended virtual synchrony, and the objective of [14] is to show how this can be used to obtain virtual synchrony as implemented in Flush Spread. Finally, [1] shows how Secure Spread, a layer for secure group communication, can be built on top of Flush Spread and the Cliques toolkit [17, 4], a C library (see http://sconce.ics.uci.edu/cliques/) that supports group key agreement and is to a large degree independent of the group communication system.

A formal executable model of Secure Spread was developed in Maude. The model is modular with independently executable and testable components formalizing the two layers of Spread, the Flush Spread VS extension, the API of the CLIQUES cryptographic service, and Secure Spread. Sources used for developing these models included, in addition to the documents cited above, a user's guide [3], the Spread sources and online API documentation [15, 16], and many helpful discussions with the Spread group at JHU. An important goal was to obtain a mathematically satisfactory and concise description that abstracts from implementation specific aspects and covers the most general behavior that an application of Spread can observe. The resulting specification serves as precise documentation that models the behavior intended to be observable and clarifies the virtual synchrony and ordering guarantees provided by each component. In addition, the suite of formal models should serve as a tool for testing alternative algorithm designs, semantic guarantees, and extensions in functionality. A future possibility is to use this model as the basis for the replicated database application built on Spread.

A number of issues arose in developing the models. These are briefly indicated below. Details can be found in the "Lessons Learned" document on the case study Web site.

Configuration Layer. At the configuration layer we needed to generalize the conditions of [2] to include all message types, since [2] covers only three delivery modes—namely, causal, agreed, and safe, but not reliable and fifo. The causal order delivery conditions of [2] conflict with self-delivery if generalized to include other message types

Group Layer. In addition to messages discussed in [14], the Spread group layer API provides group join and leave messages and client connect and disconnect (the latter entailing change in all groups of which the client is a member). We had to rely on our understanding of the source code to determine the semantics of these messages. We also defined a notion of private group to model unicast, needed for Secure Spread.

Flush Spread. The specification of Flush Spread relied mainly on [14]. Both the implementation and modeling of Flush Spread were made easier by the clearly spelled-out API for Spread.

Cliques. Our goal for this component was to come up with a formal specification that exactly captures what has been implemented for the capabilities of the Cliques Toolkit used in Secure Spread, and also captures the generic nature of the Cliques API so that further capabilities can be added without major modifications. The available documentation was somewhat outdated and we had to rely to a large degree on understanding of the C code. The result is a rather abstract but realistic specification.

Secure Spread. (seehttp://www.cnds.jhu.edu/research/group/secure_spread/) Provides the glue between Flush Spread and the Cliques toolkit. We specified the basic algorithm, formulated as a state machine, as presented in [1]. A few gaps had to be filled in and small errors corrected. For example, treatment of a join was not clear in the state machine specification. To maintain a consistent state at the group layer in our specification, events are handled even when the state machine is not yet running for a particular group (the client has not joined). The state machine description also makes an illegal call to the Cliques API. Our specification provides one solution, and the implementers have provided another. Also, some assignments in the state machine description should be Cliques procedure calls.

Since we built the specification of Secure Spread on well-tested building blocks—namely, the Flush Spread and the Cliques specifications—the effort that we spend on this layer was very reasonable. We expect that it would be very easy to optimize the algorithm as discussed in [1], because all the necessary subprotocols are provided by the Cliques specification.

Since Spread accounts for process failure and network partitions as well as asynchronous application input, both the system and its formal model are highly nondeterministic. To manage the complexity of analysis in the presence of massive nondeterminism, we developed the notion of controller process that can be composed with an initial system configuration to constrain the reachable state space and to drive the execution to specific situations of interest, that is, to constrain execution to an abstract scenario of interest. This provides a useful form of analysis that gives broader coverage than testing and can be applied to larger initial configurations than unconstrained search or model-checking. We have defined a simple controller language based on sequential and parallel composition of actions. The actions directly correspond to the rewrite rules of our system that we would like to control.

In addition to controller-based analysis, we defined variants of the published axioms for group communication systems [5, 14] as properties of the Maude models for the configuration and group layers and sketched proofs that the Maude specification satisfies relevant axioms. The process of determining the relevant and desired properties for each layer and working out proofs raised a number of issues regarding the intended behavior of Spread from different points of view, and exposed some problems with the formal specification that were subsequently fixed.

Although [2] clearly documents the configuration layer, and [14] is intended to capture the group layer, it seemed meaningful to reinterpret [14] and apply it to the configuration level (thinking of a configuration as a single physical group). If we interpret send from the Spread daemons point of view, the Sending View Delivery property holds and from that Same View Delivery for send interpreted from the client application's point of view can be inferred. The conflict between Self Delivery and causal order delivery was discovered while we carried out this validation of the configuration layer. Generalizing these properties to the case in which view changes are caused not only by network change but also by joins, leaves, and disconnects unearthed new problems with the Same View Delivery property, namely, join/leave notifications are implemented as agreed messages, but reliable, fifo, and causal messages can be delivered before or after depending on the process. Hence, Same View Delivery can be violated for messages lower than agreed. The modularity and compositionality techniques used in developing the Secure Spread specification were essential to managing the complexity inherent in developing faithful specifications of real systems. The benefits of this approach include:

- Understanding of each component in isolation
- Modular testing and analysis
- Modular structure of specification leading to a modular structure of proofs
- Allowing parallel development and collaboration.

The models, analyses, reinterpreted GCS properties, informal proofs, and supporting documentation can be found at http://formal.cs.uiuc.edu/stehr/spread_eng.html (also accessible from the project Web site).

5.0 Mobile Maude

Mobile Maude is a Mobile Agent language extending the rewriting logic [11] language Maude [7, 6] and supporting mobile computation. Mobile Maude uses reflection to obtain a simple and general declarative mobile language design and makes possible strong assurances of mobile agent behavior. The two key notions are processes and mobile objects. Processes are located in computational environments where mobile objects can reside. Mobile objects have their own code, can move between different processes in different locations, and can communicate asynchronously with each other by means of messages.

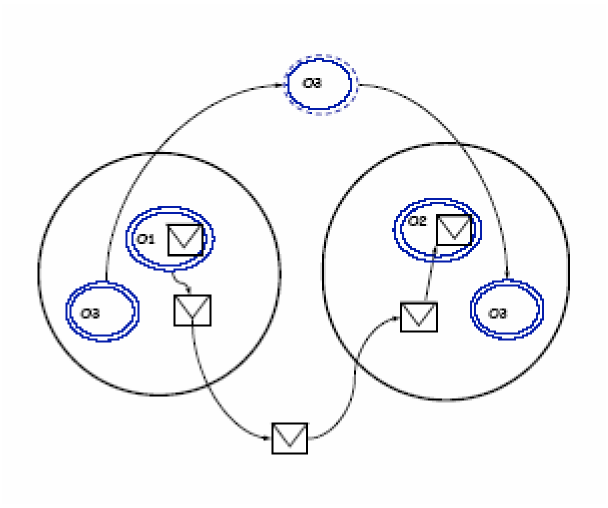


Figure 4: Mobile Maude: object and message mobility

Figure 4 illustrates a prototypical Mobile Maude system with two located processes and three mobile agents, where agent 03 is moving from one location to another. In addition, agent 01 is sending a message to agent 02. Each mobile object is created by some process, and that process is responsible for keeping track of the location of the mobile object in order that messages may be delivered.

The Mobile Maude prototype consists of a collection of Maude modules specifying infrastructure to support modeling, prototyping, and analysis of mobile agents. The initial design of Mobile Maude was presented in [9]. The implementation is founded on the Russian Dolls reflective distributed object framework [12] and the IMaude environment [10] for interacting with Maude specifications. The communication infrastructure for Mobile Maude is provided by the IOP interoperability platform [10].

In the Russian Dolls framework an object has an identifier, a class identifier, a set of attributes (fields), and an interface. The behavior of an object is specified by a set of rewrite rules associated to the objects class. Objects may be nested, hence the Russian Dolls metaphor. The object's interface makes explicit its points of interaction with its environment, both its peers and its containing object, if any. In Mobile Maude an interface consists of two message queues, one for incoming messages and one for outgoing messages.

Formally, a located process (briefly a process) is an object whose attributes include a configuration of mobile objects and messages, and a mapping from identifiers of objects it has created to the object's last known location and the number of hops the object took to reach its last location. Each time a mobile object arrives at a new location (as the contents of a message) the receiving process notifies the object's creator of the new location and hop count. The hop count is used to identify stale location information. A process also keeps information to allow it to generate a new identifier each time it creates a mobile object.

A mobile object carries its code and state with it when it moves to a new location. The code for a Maude object is the module containing the declarations for the object attributes and the rewrite rules specifying its behavior. Reflection is used to transform code and state into data that can be understood by any located process. In particular, a mobile object has two levels (indicated by the double oval in Figure 4): a meta level (called the mobile object wrapper), and a base level (called the base object or base agent). A mobile object wrapper has the same attributes for all mobile objects. These consist of metarepresentations of the base object and of the module specifying the base object behavior, an integer recording the number of hops that the object has made (initially 0), a bound on the number of rewrites the object can use in any execution step, and auxiliary data used to keep track of the wrapper's processing state. The bound on rewrites is a form of resource management and is used to ensure that all mobile objects in a given location will have a fair share of the available resources.

The rules for located process behavior specify how messages output by its mobile objects are processed, and how it processes messages from peer located objects. The interesting mobile object message is the go message which specifies the target location as well as the data needed to reconstitute the object at the new location. This is turned into an install request addressed to the peer process location, and dually when a process receives an install request it reconstitutes the mobile object in its configuration and reports the new location. In addition, processes send and receive requests to route messages to mobile objects. These are handled using the object location mapping.

The rules for the mobile object wrapper specify how the wrapper manages its base object. Messages sent by the base level to peers are forwarded to the peer wrapper for delivery, and messages for delivery received from peer wrappers are delivered to the base object, using the reflected evaluation and rewriting functions. Requests for new object creation are modified by adding a mobile object wrapper to the object specification. Mobility for a base object means that it can send messages of the form go(loc). To process such a message, the mobile wrapper packages its identifier, class id, attributes (which include the base object and code module), and interface queues and turns itself into a request to go to the location specified by the base object, containing the packaged information.

Using the Mobile Maude infrastructure, to specify a mobile agent system the user needs only to specify the behavior of base objects. These are ordinary Maude objects with the Russian Dolls object structure, specified by defining the class ids, attributes, and contents of messages sent to peers. In addition to peer-to-peer messages, a base mobile object may send new object requests (handled by the container process) and go requests.

The Mobile Maude infrastructure is designed with independently executable layers to allow incremental testing and validation of mobile agent specifications. A system of base objects may be executed by importing a module that defines base mobile object system composition by adding necessary communication rules. Here, location is simply in the mind of the object and the object marshaling and unmarshaling operations are not needed. A configuration of mobile wrapper objects can be executed by importing a module that defines wrapped mobile object system composition and communication rules. Finally, a collection of located processes can be executed by importing a module that defines process communication rules. This can all be done by using the Maude command line interface. This approach has the limitation that interaction with the mobile system cannot be controlled, and all messages to be considered must be present in the initial configuration. By using an IMaude execution environment the user can start a system, and incrementally send messages and receive replies, giving a more controlled test environment. This environment is included in the Mobile Maude implementation package.

A simple example of a data mobile that goes from place to place collecting data and answering queries is included with infrastructure modules along with example test runs to illustrate the process of defining and testing mobile object systems.

The Mobile Maude prototype, execution environments, example and documentation can befound at http://www-formal.stanford.edu/clt/FTN/MobileMaude.

6.0 References

[1] Y. Amir, Y. Kim, C. Nita-Rotaru, J. Schultz, J. Stanton, and G. Tsudik. Secure group communication using robust contributory key agreement. *IEEE Transactions on Parallel and Distributed Systems Archive*, 15(5):468–480, 2004.

[2] Yair Amir. *Replication Using Group Communication Over a Partitioned Network*. PhD thesis, Hebrew University of Jerusalem, 1995.

[3] Yair Amir and Jonathan Stanton. A user’s guide to Spread, October 2002. Available from the Spread Web site http://www.spread.org.

[4] G. Ateniese, O. Chevassut, D. Hasse, Y. Kim, and G. Tsudik. The design of a group key agreement API. In *DARPA Information Survivability Conference and Exposition (DISCEX 2000)*. IEEEE, January 2000.

[5] Gregory V. Chockler, Idit Keidar, and Roman Vitenberg. Group communication specifications: A comprehensive study. *ACM Computing Surveys*, 33(4):1–43, December 2001.

[6] M. Clavel, F. Dur´an, S. Eker, P. Lincoln, N. Mart´ı-Oliet, J. Meseguer, and J. Quesada. Maude: Specification and programming in Rewriting Logic. *Theoretical Computer Science*, 285(2), 2002.

[7] M. Clavel, F. Dur´an, S. Eker, P. Lincoln, N. Marti-Oliet, J. Meseguer, and C. Talcott. Maude 2.0 Manual, 2003. http://maude.cs.uiuc.edu.

[8] Manuel Clavel and Jos´e Meseguer. Reflection in Conditional Rewriting Logic. *Theoretical Computer Science*, 285:245–288, 2002.

[9] Francisco Dur´an, Steven Eker, Patrick Lincoln, and Jos´e Meseguer. Principles of Mobile Maude. In *Agent Systems, Mobile Agents, and Applications, ASA/MA 2000*, volume 1882 of *Lecture Notes in Computer Science*, pages 73–85. Springer-Verlag, 2000.

[10] I. A. Mason and C. L. Talcott. IOP: The InterOperability Platform & IMaude: An interactive extension of Maude. In *Fifth International Workshop on Rewriting Logic and Its Applications (WRLA'2004)*, Electronic Notes in Theoretical Computer Science. Elsevier, 2004.

[11] J. Meseguer. Conditional Rewriting Logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[12] J. Meseguer and C. L. Talcott. Semantic models for distributed object reflection. In *European Conference on Object-Oriented Programming, ECOOP'2002*, volume 2374 of *Lecture Notes in Computer Science*, pages 1–36, 2002. invited paper.

[13] Jos´e Meseguer. A logical theory of concurrent objects and its realization in the Maude language. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 314–390. MIT Press, 1993.

[14] John Schultz. Partitionable virtual synchrony using extended virtual synchrony. Master's thesis, The Johns Hopkins University, 2001.

[15] Spread souce code. version 2.1.0. Available at www.spread.org.

[16] Spread souce code. version 2.17.2rc3. Available at www.spread.org.

[17] M. Steiner, G. Tsudik, and M. Waidner. Key agreement in dynamic peer groups. *IEEE Transactions on Parallel and Distributed Systems*, August 2000.